TLB Operations Based on Shared Bit

Gerard Chauvel

This application claims priority to European Application Serial No. 00402331.3, filed August 21, 2000 (TI-31366EU) and to European Application Serial No. 01401218.1, filed May 11, 2001 (TI-31357EU). US Patent Application Serial No. _____ (TI-31366US) is incorporated herein by reference.

Co-Related Applications

This application is related to patent application Serial No. _____(TI-31355), entitled *TLB Operation Based on Task-ID*; patent application Serial No. _____ (TI-31356), entitled *TLB Lock and Unlock Operation*; and patent application Serial No. _____ (TI-31358), entitled *TLB with Resource ID Field*.

Field of the Invention

This invention generally relates to computer processors, and more specifically to improvements in translation lookaside buffers for address translation, systems, and methods of making.

Background

Microprocessors are general-purpose processors that provide high instruction throughputs in order to execute software running thereon, and can have a wide range of processing requirements depending on the particular software applications involved.

Many different types of processors are known, of which microprocessors are but one example. For example, Digital Signal Processors (DSPs) are widely used, in particular for specific applications, such as mobile processing applications. DSPs are typically configured to optimize the performance of the applications concerned and to achieve this they employ more specialized execution units and instruction sets. Particularly in applications such as mobile telecommunications, but not exclusively, it is desirable to provide ever increasing DSP performance while keeping power consumption as low as possible.

To further improve performance of a digital system, two or more processors can be interconnected. For example, a DSP may be interconnected with a general purpose processor in a digital system. The DSP performs numeric intensive signal processing algorithms while the general purpose processor manages overall control flow. The two processors communicate and transfer data for signal processing via shared memory. A direct memory access (DMA) controller is often associated with a processor in order to take over the burden of transferring blocks of data from one memory or peripheral resource to another and to thereby improve the performance of the processor.

Modular programming builds a computer program by combining independently executable units of computer code (known as modules), and by tying modules together with additional computer code. Features and functionality that may not be provided by a single module may be added to a computer program by using additional modules.

The design of a computer programming unit known as a task (or function) is often accomplished through modular programming, where a specific task is comprised of one module and the additional computer code needed to complete the task (if any additional code is needed). However, a task may be defined as broadly as a grouping of modules and additional computer codes, or, as narrowly as a single assembly-type stepwise command. A computer program may be processed (also called "run" or "executed") in a variety of manners. One manner is to process the

computer code sequentially, as the computer code appears on a written page or on a computer screen, one command at a time. An alternative manner of processing computer code is called task processing. In task processing, a computer may process computer code one task at a time, or may process multiple tasks simultaneously. In any event, when processing tasks, it is generally beneficial to process tasks in some optimal order.

Unfortunately, different tasks take different amounts of time to process. In addition, the result, output, or end point of one task may be required before a second task may begin (or complete) processing. Furthermore, particularly in a multiple processor environment, several tasks may need access to a common resource that has a generally fixed capacity.

In order to better manage program tasks and physical memory, a concept of virtual memory and physical memory has evolved. Program task modules are generally compiled and referenced to virtual address. When a task is executed in physical memory, address translation is performed using a cache of translated addresses, referred to as a translation lookaside buffer (TLB). TLBs must be managed to optimize system performance as various tasks are executed.

Accordingly, there is needed a system and method for managing task processing and address translation that takes into account active tasks, active resources, and other task processing needs.

Summary of the Invention

Particular and preferred aspects of the invention are set out in the accompanying independent and dependent claims. In accordance with a first embodiment of the invention, a method is provided for operating a digital system having a processor and associated translation lookaside buffer (TLB). Several program tasks are executed within the processor that initiate a sequence of memory access requests in response to the program tasks. In response to the sequence of memory access requests, a set of translated memory addresses are cached in the TLB. A task identification value is incorporated with each translated memory address to identify which of the program tasks requested the respective translated memory address. A shared indicator is also incorporated with each translated memory address to indicate when a translated memory address is shared by more than one of the program tasks. A portion of the plurality of translated memory address in the TLB are invalidated in a manner that is qualified by the shared indicator. In one case, only shared translated addresses are invalidated. In another case, only translated address that are not shared are invalidated.

In another embodiment, the TLB has several levels, and the step of invalidating encompasses all of the levels of the TLB.

In another embodiment, a set of page translation tables is maintained for providing each translated memory address. The shared indicator for each translated memory address is included within the set of page translation tables. In another embodiment, the shared indicator is not included within the set of page tables but within other specific tables associated to each task and managed by the OS.

In another embodiment, a shared translated memory address is replicated in the TLB for each program task that requests it.

In another embodiment, the program tasks exist in a single address space, and only a single copy of a shared translated memory address is cached in the TLB.

Another embodiment of the invention is a digital system that has a translation lookaside buffer (TLB). The TLB includes storage circuitry with a set of entry locations for holding translated values, wherein each of the set of entry locations includes a first field for a translated value and a second field for an associated shared indicator. There is a set of inputs for receiving a translation request, a set of outputs for providing a translated value selected from the set of entry locations; and control circuitry connected to the storage circuitry. The control circuitry is responsive to an operation command to invalidate selected ones of the set of entry locations according to the shared indicator.

Brief Description of the Drawings

Particular embodiments in accordance with the invention will now be described, by way of example only, and with reference to the accompanying drawings in which like reference signs are used to denote like parts and in which the Figures relate to the digital system of Figure 1 and in which:

Figure 1 is a block diagram of a digital system that includes an embodiment of the present invention in a megacell core having multiple processor cores;

Figure 2A and 2B together is a more detailed block diagram of the megacell core of Figure 1;

Figure 3A is a block diagram illustrating a shared translation lookaside buffer (TLB) and several associated micro-TLBs (µTLB) included in the megacell of Figure 2;

Figure 3B is a flow chart illustrating a method of operating the TLB of Figure 3A;

Figure 4 is a block diagram of a digital system similar to Figure 1 illustrating a cloud of tasks that are scheduled for execution on the various processors of the digital system;

Figure 5 illustrates a TLB control format used to operate on the TLB and µTLBs of Figure 3A;

Figure 6 illustrates operation of the TLB of Figure 3A for selective flushing of an entry for a given task or resource;

Figure 7 illustrates control circuitry for adaptive replacement of TLB entries in the TLB of Figure 3A;

Figure 8 illustrates how a shared page entry is replicated for each task for different virtual address spaces;

Figure 9 illustrates how a shared page entry is used by each of the sharing tasks in a single virtual address space; and

Figure 10 is a representation of a telecommunications device incorporating an embodiment of the present invention.

TI-31357   - 6 -

Corresponding numerals and symbols in the different figures and tables refer to corresponding parts unless otherwise indicated.

Detailed Description of Embodiments of the Invention

[01]   Although the invention finds particular application to Digital Signal Processors (DSPs), implemented, for example, in an Application Specific Integrated Circuit (ASIC), it also finds application to other forms of processors. An ASIC may contain one or more megacells which each include custom designed functional circuits combined with pre-designed functional circuits provided by a design library.

[02]   Figure 1 is a block diagram of a digital system that includes an embodiment of the present invention in a megacell core 100 having multiple processor cores. In the interest of clarity, Figure 1 only shows those portions of megacell 100 that are relevant to an understanding of an embodiment of the present invention. Details of general construction for DSPs are well known, and may be found readily elsewhere. For example, U.S. Patent 5,072,418 issued to Frederick Boutaud, et al, describes a DSP in detail. U.S. Patent 5,329,471 issued to Gary Swoboda, et al, describes in detail how to test and emulate a DSP. Details of portions of megacell 100 relevant to an embodiment of the present invention are explained in sufficient detail herein below, so as to enable one of ordinary skill in the microprocessor art to make and use the invention.

[03]   Referring again to Figure 1, megacell 100 includes a control processor (MPU) 102 with a 32-bit core 103 and a digital signal processor (DSP) 104 with a DSP core 105 that share a block of memory 113 and a cache 114, that are referred to as a level two (L2) memory subsystem 112. A traffic control block 110 receives transfer requests from a host processor connected to host interface 120b, requests from control processor 102, and transfer requests from a memory access node in DSP 104. The traffic control block interleaves these requests and presents them to the shared memory and cache. Shared peripherals 116 are also accessed via the traffic control block. A direct memory access controller 106 can transfer data between an external source such as off-chip memory 132 or on-chip memory 134 and the shared memory. Various application specific processors or hardware accelerators

108 can also be included within the megacell as required for various applications and interact with the DSP and MPU via the traffic control block.

[04] External to the megacell, a level three (L3) control block 130 is connected to receive memory requests from internal traffic control block 110 in response to explicit requests from the DSP or MPU, or from misses in shared cache 114. Off chip external memory 132 and/or on-chip memory 134 is connected to system traffic controller 130; these are referred to as L3 memory subsystems. A frame buffer 136 and a display device 138 are connected to the system traffic controller to receive data for displaying graphical images. A host processor 120a interacts with the external resources through system traffic controller 130. A host interface connected to traffic controller 130 allows access by host 120a to external memories and other devices connected to traffic controller 130. Thus, a host processor can be connected at level three or at level two in various embodiments. A set of private peripherals 140 are connected to the DSP, while another set of private peripherals 142 are connected to the MPU.

[05] Figure 2, comprised of Figure 2A Figure 2B together, is a more detailed block diagram of the megacell core of Figure 1. DSP 104 includes a configurable cache 203 that is configured as a local memory 200 and data cache 202, and a configurable cache 204 that is configured as instruction cache 206 and a RAM-set 208, which are referred to as level one (L1) memory subsystems. The DSP is connected to the traffic controller via an L2 interface 210 that also includes a translation lookaside buffer (TLB) 212. A DMA circuit 214 is also included within the DSP. Individual micro TLBs (µTLB) 216-218 are associated with the DMA circuit, data cache and instruction cache, respectively.

[06] Similarly, MPU 102 includes a configurable cache 223 that is configured as a local memory 220 and data cache 222, and a configurable cache 224 that is configured as instruction cache 226 and a RAM-set 228, again referred to as L1 memory subsystems. The MPU is connected to traffic controller 110 via an L2 interface 230 that also includes a TLB 232. A DMA circuit 234 is also included

within the MPU. Individual micro TLBs (μTLB) 236-238 are associated with the DMA circuit, data cache and instruction cache, respectively.

[07]   L2 traffic controller 110 includes a TLB 240 and one or more micro-TLB (μTLB) 242 that are associated with system DMA block 106, host processor interface 120b for a host connected at level two, and other application specific hardware accelerator blocks. Similarly, L3 traffic controller 130 includes a μTLB controllably connected to TLB 240 that is associated with system host 120a at level three. This μTLB is likewise controlled by one of the megacell 100 processors.

Memory Management Unit

[08]   At the megacell traffic controller level, all addresses are physical. They have been translated from virtual to physical at the processor sub-system level by a memory management unit (MMU) associated with each core, such as DSP core 105 and MPU core 103. At the processor level, access permission, supplied through MMU page descriptors, is also checked, while at the megacell level protection between processors is enforced by others means, which will be described in more detail later. Each MMU includes a TLB and its associated μTLBs.

[09]   The translation lookaside buffer (TLB) caches contain entries for virtual-to-physical address translation and page descriptor information such as access permission checking, cache policy for various levels, etc. If the TLB contains a translated entry for the virtual address, the access control logic determines whether the access is permitted. If access is permitted, the MMU generates the appropriate physical address corresponding to the virtual address. If access is not permitted, the MMU sends an abort signal via signal group 244 to the master CPU 102. The master CPU is identified by the value of the R-ID field. On a slave processor such as a hardware accelerator the R-ID is equal to the R-ID of the master CPU.

[10]   Upon a TLB miss, i.e., the TLB does not contain an entry corresponding to the virtual address requested, an exception is generated that

initiates a translation table walk software routine. The TLB miss software handler retrieves the translation and access permission information from a translation table in physical memory. Once retrieved, the page or section descriptor is stored into the TLB at a selected victim location. Victim location selection is done by software or with hardware support, as will be described later.

Translation Table

[11]   To provide maximum flexibility, the MMU is implemented as a software table walk, backed up by TLB caches both at the processor sub-system and megacell level. This allows easy addition of new page size support or new page descriptor information if required. A TLB miss initiates a TLB handler routine to load the missing reference into the TLB. At the Megacell 100 level, a TLB miss asserts a miss signal in signal group 244 and is routed via system interrupt router 250 to the processor having generated the missing reference or to the processor in charge of the global memory management, via interrupt signals 251, 252.

[12]   Translation tables and TLB cache contents must be kept consistent. A flush operation is provided for this reason and will be described in more detail later.

[13]   An address reference is generally located within the μTLB or main TLB of each processor sub-system; however, certain references, such as those used by system DMA 106 or host processor 120, for example, to access megacell memories can be distributed within L2 traffic controller 110 and cached into L2 system shared TLB 240. Because system performance is very sensitive to the TLB architecture and size, it is important to implement efficient TLB control commands to lock entries for critical tasks or unlock and flush those entries when a task is deleted without degrading the execution of other tasks. Therefore, each TLB and L2 cache entry holds a task-ID. Commands are supplied to flush locked or unlocked entries of a TLB/μTLB corresponding to a selected task.

[14]   As part of the page descriptor information, the MMU provides cacheability and bufferability attributes for all levels of memory. The MMU also

provides a "Shared" bit for each entry to indicate that a page is shared among multiple processors (or tasks). This bit, as standalone or combined with the task-ID, allows specific cache and TLB operation on data shared between processors or/and tasks. The MMU may also provide additional information, such as memory access permission and access priority as described later.

[15]   All megacell memory accesses are protected by a TLB. As they all have different requirements in term of access frequencies and memory size, a shared TLB with individual μTLB backup approach has been chosen to reduce the system cost at the megacell level. This shared TLB is programmable by each processor. The architecture provides enough flexibility to let the platform work with either independent operating systems (OS) on each processors or a distributed OS with a unified memory management, for example.

[16]   The present embodiment has a distributed operating system (OS) with several domains corresponding to each processor but only a single table manager for all processors. Slave processors do not manage the tables. In a first embodiment slave processors R-ID are equal to the R-ID of the master CPU. In another embodiment, they could, however, have a different R-ID to control their TLB entries lock/unlock entries corresponding to some of their own tasks or flush all their entries, when putting themselves in sleep mode to free entries for the others processors. Having different R-ID provides a means to increase security in a concurrent multi-processor environment, processor X can not access memory allocated to processor Y.

[17]   In another embodiment with several independent OS(s), for example, there will be independent tables. These tables can be located in a memory space only viewed by the OS that they are associated with in order to provide protection from inadvertent modification by another OS. As they manage the virtual memory and task independently, the R-ID provides the necessary inter-processor security. R-Ids are managed by a single master CPU. This CPU can make TLB operations on all TLB entries. TLB operation or memory accesses from slave processor are

restricted by their own R-ID. The CPU master will have rights to flush out entries belonging to another processor in a different OS domain.

[18]   The organization of the data structures supporting the memory management descriptor is flexible since each TLB miss is resolved by a software TLB-miss handler. These data structures include the virtual-to-physical address translation and all additional descriptors to manage the memory hierarchy. A list of these descriptors and their function is described in Table 2. Table 1 includes a set of memory access permission attributes, as an example. In other embodiments, a processor may have other modes that enable access to memory without permission checks.

Table 1 - Memory Access Permission

Supervisor	User
No access	No access
Read only	No access
Read only	Read only
Read/Write	No access
Read/Write	Read only
Read/Write	Read/Write

Table 2 – Memory Management Descriptors

Execute Never	provides access permission to protect data memory area from being executed. This information can be combined with the access permission described above or kept separate.
Shared	indicates that this page may be shared by multiple tasks across multiple processor.
Cacheability	Various memory entities such as individual processor's cache and write buffer, and shared cache and write buffer are managed through the MMU descriptor. The options included in the present embodiment are as follows: Inner cacheable, Outer cacheable, Inner Write through/write back, Outer write through/write back, and Outer write allocate. The terms Inner and outer refer to levels of caches that are be built in the system. The boundary between inner and outer is defined in specific embodiment, but inner will always include L1 cache. In a system with 3 levels of caches, the inner correspond to L1 and L2 cache and the outer correspond to L3 due to existing processor systems. In the present embodiment, inner is L1 and outer is L2 cache.
Endianism	determines on a page basis the endianness of the transfer.
priority	Indicates a priority level for the associated memory address region. Memory access can be prioritized based on this priority value.

MMU/TLB Control Operation

[19]   Figure 3A is a block diagram illustrating a shared translation look-aside buffer (TLB) 300 and several associated micro-TLBs (µTLB) 310(0)-310(m) included in megacell 100 of Figure 2. On a µTLB miss, the shared TLB is first searched. TLB controller 320 is alerted by asserting a µTLB miss signal 324. In case of a hit on the shared TLB, the µTLB that missed is loaded with the entry content of the shared TLB 300. In the case of a miss in shared TLB 300, the shared TLB alerts TLB controller 320 by asserting a TLB miss signal 326. Controller 320 then asserts an interrupt request signal 328 to system interrupt controller 250. Interrupt controller 250 asserts an interrupt to the processor whose OS supervises the resource which caused the miss. A TLB entry register 330 associated with TLB controller 320 is loaded by a software TLB handler in response to the interrupt. Once loaded, the contents of TLB entry register 330 are transferred to both shared TLB 300 and the requesting µTLB at a selected victim location as indicated by arcs 332 and 334.

[20]   A separate TLB entry register 330 is only one possible implementation and is not necessarily required. The separate register TLB entry register is a

memory mapped register that allows buffering of a complete TLB entry (more than 32 bits). A TLB value is not written directly in the TLB cache but is written to the TLB entry register first. Because of the size of an entry, several writes are required to load the TLB entry register. Loading of a TLB cache entry is then done in a single operation "Write TLB entry". Advantageously, other uTLBs associated with other modules can continue to access the shared TLB while the TLB entry register is being loaded, until a second miss occurs. Advantageously, by controlling access to the TLB via the TLB entry register, CPUs have no direct access to TLB cache internal structure and thus the risk of partial modifications inconsistent with the MMU tables is avoided.

[21]   The sequence of operations to update a TLB cache entry after a miss is:

1 - the software TLB handler writes to the TLB entry register,

2- the software TLB handler sends a command to write the TLB entry, which transfers a value from TLB entry register to a preselected victim TLB cache entry; and

3- control circuitry checks and preselects a next victim TLB entry, in preparation for the next miss. In this embodiment, this step is generally performed in background prior to the occurrence of a miss.

[22]   Advantageously, TLB cache entries can be preemptively updated under OS software control to prevent TLB miss by pre-loading a new entry, using the following sequence of operation:

1- control circuitry checks and selects a TLB entry, referred to as a victim TLB cache entry.

2- the software TLB handler writes to the TLB entry register, and

3- the software TLB handler sends a command to write the TLB entry, which transfers a value from TLB entry register to the selected victim TLB cache entry.

[23]   The priority on the shared TLB is managed in the same way as priority on a memory access. One or more resources can be using the shared TLB. One or more

resources can program the shared TLB. The replacement algorithm for selecting the next victim location in the shared TLB is under hardware control. A victim pointer register 322 is maintained for each TLB and μTLB to provide a victim separate pointer for each. A typical embodiment will use a round robin scheme. Different TLBs within a single megacell can use different replacement schemes. However, in an embodiment in which the system has a master CPU with a distributed OS, this master CPU could also bypass the hardware replacement algorithm by selecting a victim entry, reading and then writing directly to the Shared TLB, for example.

[24]   In this embodiment, each shared TLB has 256 entries. Each μTLB is generally much smaller, i.e., has fewer entries, than the shared TLB. In various embodiments, each shared TLB has 64-256 or more entries while μTLBs generally have 4-16 entries. The penalty for a miss in a μTLB is small since a correct entry is generally available from the shared TLB. Therefore, the present embodiment does not provide direct control of the victim pointers of the various μTLBs; however, direct control of the victim pointer of shared TLBs, such as 212, 232, and 240, is provided.

[25]   Each entry in a TLB has a resource identifier 301 along with task-ID 302. Resource-IDs and task IDs are not extension fields of the virtual address (VA) but simply address qualifiers. Resource IDs are provided by a resource-ID register associated with each resource; such as R-ID register 342a associated with resource 340 and R-ID register 342n associated with resource 350. Resource 340 is representative of various DMA engines, coprocessor, etc within megacell 100 and/or an external host connected to megacell 100. Resource 350 is representative of various processors within megacell 100. Each resource 340, 350 typically has its own associated R-ID register; however, various embodiments may choose to provide resource ID registers for only a selected portion of the resources. A task ID is provided by a task-ID register, such as task-ID register 344a associated with resource 340 and task-ID register 344n associated with resource 350. A task

register associated with a non-processor resource, such as DMA, a coprocessor, etc, is loaded with a task value to indicate the task that it is supporting.

[26]   In another embodiment, only processor resources 340, 350 that execute program modules have an associated programmable task-ID register. In this case, a system wide default value may be provided for access requests initiated by non-processor resources such as DMA. The default value may be provided by a programmable register or hardwired bus keepers, for example.

[27]   Advantageously, with the task-ID, all entries in a TLB belonging to a specific task can be identified. They can, for instance, be invalidated altogether through a single operation without affecting the other tasks. Advantageously, the resource ID permits discrimination of different tasks being executed on different resources when they have the same task number. Task-ID number on the different processors might not be related; therefore, task related operations must be, in some cases, qualified by a resource-ID.

[28]   In another embodiment, the R-ID and Task_ID registers are not necessarily part of the resource core and can be located elsewhere in the system, such as a memory mapped register for example, and associated to a resource bus. The only constraint is that a task_ID register related to a CPU must be under the associated OS control and updated during context switch. R-ID must be set during the system initialization. In some embodiments at system initialization, all R-ID and Task-ID registers distributed across the system are set to zero, which is a default value that causes the field to be ignored. In other embodiments, a different default value may be used. In other embodiments, R-ID "registers" provide hardwired values.

[29]   Referring still to Figure 3A, each TLB entry includes a virtual address field 305 and a corresponding physical address field 308 and address attributes 309. Various address attributes are described in Table 1 and Table 2. Address attributes define conditions or states that apply to an entire section or page of the address space that is represented by a given TLB entry. An S/P field 306 specifies a

page size such as 64kB and 4kB for example. Naturally, the page size determines how many most significant (ms) address bits are included in a check for an entry.

[30]   Each TLB entry also includes "shared" bit 303 and a lock bit 304. All entries marked as shared can be flushed in one cycle globally. A V field 307 indicates if an associated TLB cache entry is valid. V field 307 includes several V-bits that are respectively associated with R-ID field 301 to indicate if a valid R-ID entry is present, task-ID field 302 to indicate if a valid task-ID entry is present, and virtual address field 305 to indicate if a valid address entry is present. These valid bits enable the compare logic with their associated field.

[31]   As mentioned earlier, the resource ID field and task ID field in each entry of the TLB/μTLB can be used to improve security. During program task execution, each transaction request is checked by the miss control circuitry of the TLB/μTLB to determine if the entry is allowed for a specific resource or for all resources and for a specific task or for all tasks. For example, if a request is received and a valid entry is present for the proffered virtual address but a task ID or R-ID which accompany the request does not match the corresponding valid task ID and R-ID fields of the entry, then a miss is declared. If the task ID and/or R-ID fields of the entry are marked as invalid, then they are ignored.

[32]   Figure 3B is a flow chart illustrating a method of operating the TLB of Figure 3A. As discussed above, the TLB is filled during the normal course of action by a set of translated address entries along with qualifier fields that are incorporated with each entry. As will be described in more detail below, operations can now be performed on the TLB that are qualified by the various qualifier fields.

[33]   In step 360, an operation command is received by the control circuitry of the TLB. This command is sent by the MMU manager during the course of operation. Commands are sent as needed to flush (invalidate), lock or unlock selected entries within the TLB. These operations will be described in detail later.

[34]   Step 362 accesses a first entry in the TLB and reads the qualifier field specified by the operation command. This can be task ID field 302, resource ID field

301, shared indicator 303, or combinations of these. Operation commands can also specify a selected virtual address entry.

[35]   Step 364 compares the qualifier specified by the operation command with the qualifier field read from the TLB entry. If they match, then the operation is performed on that entry in step 366. If they do not match, then the next entry is accessed in step 368 and compare step 364 is repeated for the next entry.

[36]   Step 366 performs the operation specified in the operation command on each entry whose qualifier field(s) match the operation command. In this embodiment, the operation can invalidate an entry by resetting valid bit field 307, and lock or unlock an entry by appropriate setting of lock bit 304.

[37]   Step 368 access each next TLB entry until all entries have been accessed. In this embodiment, all μTLBs associated with a shared TLB are also accessed as part of the same operation command.

[38]   Other embodiments may provide additional or different operations that are qualified by the qualifier fields of the present embodiment or by additional or other types of qualifier fields. For example, resource type, power consumption, processor speed, instruction set family, and the like may be incorporated in the TLB and used to qualify operations on the TLB.

[39]   Figure 4 is a block diagram of a digital system similar to that of Figure 1 illustrating cloud of tasks that are scheduled for execution on the various processors of the digital system. Typically, each software task includes a task priority value that is commonly used by an operating system to schedule an order of execution for a set of pending tasks 1440.

[40]   In this illustration, a circle such as 1442 represents a task, with a task name "c" and a task priority of 12, for example. Likewise, task 1443 has a task name "r" and a priority of 15, where a lower number indicates a higher priority. If the set of tasks 1440 are assigned to three processors, then an operating system on each processor forms a ready to execute queue, such as ready queue 1446 in which task "c" is scheduled for first execution, then task "a" and finally task "b" according

to priority values of 12, 15, and 50 respectively. The Task ID register in each processor is loaded when a task is invoked.

[41]   Table 3 illustrates several portions of instruction code sequences in which a task is spawned. From line 1 to line 5, task "c" is active and spawns a new task, "audio" on line 5. The kernel is then invoked to instantiate the new task and create the associated TCB. An eight bit (numbers of bits can be more or less) task-ID field is memorised in the TCB at line 11. During the context switch (reschedule in line 13) before launching the "audio" task, the kernel loads task-ID register 1412 with the task-ID value held in the TCB (Table 4) or in another table. At line 14, the new task is now active.

Table 3 - Setting Task ID at the Start of a Task

1	// (Task c code execution)
2	Instruction 1
3	------
4	instruction n
5	Taskspawn("audio",200,0,5000,(FUNCPTR)audio,// (Task ccode execution: instruction n+2)
6	//(Kernel code execution)
7	-----
8	TaskCreate()
9	//(taskcreate code execution)
10	------
11	SetTaskAttributeID(TID)
12	-------
13	// Kernel reschedule code execution
14	//(Task Audio code execution)
15	Instruction 1
16	-------

[42]   Table 4 is an example task control block that is used to define a task-ID. Typically, the OS uses a 32-bit task-ID that is in fact an address that enables the OS to locate task information (TCB). At line 4, an execution priority value is defined that is used by the operating system to schedule execution of the task. At line 5, a task-ID value is defined that is used to set the task ID register when the task is instantiated.

Table 4 - Setting Task ID Using a TCB

```
1      TCB (task control block)
2      Typedef struct TCB
3      {
4        UINT  OS-priority
5        UINT  Task_ID
6          ---
7      #if CPU_FAMILY == xx
8      EXC_INFO  excinfo;
9      REG_SET   regs;
10       ...
11     #endif
12     }
```

[43]   In other embodiments, other means than a TCB may be provided for storing the task ID.

[44]   Referring again to Figure 3A, task-ID field 302 can be set in response to information provided at line 5 of the TCB illustrated in Table 4. This information can be used directly by the MMU manager when loading a new entry in TLBs. This information could also be part of the page table descriptor in the MMU page table and loaded as part of the MMU software table walk.

[45]   In the present embodiment, task-ID information is not maintained in page tables but is inserted by the TLB miss handler at the time of a TLB fault by using the task_ID value of the transaction request that caused the TLB fault. Other embodiments may use other means for setting the task-ID field in the TLB entry, such as by storing this information in a separate table or in the MMU page tables, for example. In the present embodiment the Valid bit associated with the task-ID field is loaded through the MMU table walk and is part of the MMU tables. Thus, when the TLB miss handler accesses a page table in response to a TLB miss, it queries the task-ID valid bit field of the MMU page table; if this bit field is asserted, then the TLB miss handler asserts the task-ID valid bit in the TLB entry and loads the task-ID value from the task-ID register of the requester that caused the TLB miss into task ID field 302. If the task-ID valid bit field of the MMU page table is not asserted, then the TLB miss handler deasserts the task-ID valid bit in

the TLB entry and the task-ID value from the task-ID register of the requester that caused the TLB miss is ignored.

[46]   In the present embodiment, the shared bit field 303 is loaded through the MMU table walk and is part of the MMU tables. Typically, shared pages are defined by the OS in response to semaphore commands, for example.

[47]   In another embodiment, shared bit information is not maintained in page tables but is inserted by the TLB-miss handler at the time of a TLB fault by accessing the TCB directly based on the task ID of the request that caused the fault. The TCB is located by the TLB-miss handler via a look-up table keyed to the task ID value. Other embodiments may use other means for setting the shared bit in the TLB entry by storing this information in a separate table, for example.

[48]   R-ID field 301 is set by using the R-ID of the request that caused the fault. A Master CPU could also load value in this field during the programming of a TLB entry by taking this information from the MMU tables or separate tables, for example.

[49]   Figure 5 illustrates a TLB control word format used to operate on the TLB and μTLBs of Figure 3A in response to control operations as defined in Table 5. TLB control word format 400 includes a task-ID field 402, resource-ID field 404 and virtual address field 406. Note that the virtual address field refers to a page address, therefore lsb address bits that refer within a page are not needed. In some embodiments, certain of the processors might not be allowed to invalidate entries other than their own.

[50]   As described previously, during execution of a program, the R-ID and Task-ID field comes from a register associated with a requester during each memory system access request. In a system embodiment with multi-processors with multiple independent Operating Systems (OS), the R-ID is static and indicates which of the resources is accessing a given location (address). The Task-ID indicates which of the tasks (or processes) of this resource is doing the access. The task ID is dynamic and changes on each context switch. For these systems,

restricting operations on a system TLB to the associated resource is important to optimize the main system TLB usage. Each OS controls the TLB entries it uses.

[51]   However, another system embodiment might be controlled by middleware that supports a unified task and memory management. For those, the notion of R-ID might disappear and be treated as part of the task_ID. Restriction of TLB command based on R-ID would not be necessary in those systems and the field R-ID could be re-used to extend the task-ID field. In that case, TLB control format 410 may be used in which the R_Id field is not needed. Recall that the R-ID of the requestor is provided with each transaction request, therefore control operations specified using format 410 can be confined to entries associated with the requestor.

[52]   A processor can initiate various control operations on a TLB by writing a control word conforming to appropriate format to a specific memory mapped address associated with TLB controller 320. This control word can specify a target virtual address entry and an associated task ID or an associated resource ID. Depending on the operation, unneeded fields are ignored. For example, the operation "invalidate all entries related to an R-ID" will only use the R-ID field 404. The format and type of operation can be distinguished by using different memory mapped addresses, for example. Each address corresponds to a different TLB operation. Another embodiment would be to use a different processor instruction opcode for each of the TLB operation that would drive the appropriate control signal connected to TLB controller 2232. A state machine in TLB controller 320 then executes the requested control operation. These TLB control operations are listed in Table 5. These operations are described in more detail below. For many of the operations, certain processors in an embodiment will be restricted to only affecting their own entries. This restriction is enforced by using the resource-ID signals 2106 provided with each write to TLB controller 320 as part of each memory access request.

Table 5 – TLB Control Operation

Invalidate entry with VA
Invalidate all entries related to a Task-ID
Invalidate all entries related to a R-ID
Invalidate all shared entry
Invalidate all entries of a task except shared
Invalidate All entries
Lock/UnLock entry
Lock/Unlock all entries related to a task-ID/R-ID
Read TLB entry
Write TLB entry
Check and select victim TLB entry
Set victim TLB entry

[53]   In another embodiment, the control operations can be invoked by executing an instruction that invokes a hardware or software trap response. As part of this trap response, a sequence of instructions can be executed or a control word can be written to a selected address, for example. In another embodiment, one of the processors may include instruction decoding and an internal state machine(s) to perform a TLB or Cache control operation in response to executing certain instructions which may include parameters to specify the requested operation, for example.

[54]   For an "invalidate entry" operation, a Virtual page address (VA) is provided in VA field 406 of the control word and the other fields of the control word are ignored. This generates an entry invalidate operation on the corresponding virtual address entry. Note that all processors of a given megacell embodiment might not be allowed to invalidate entries others than their own. In that case, the R-ID value from the R-ID register of the requestor is used to qualify the operation.

[55]   For an "invalidate all entries related to a task" operation, all entries corresponding to the provided task identifier are invalidated. This allows a master-processor to free space from the shared TLB by invalidating all entries of a task belonging to another processor. In this case, the control word provides a task-ID value and an R_ID value. Processors other than the master-processor can free space from the shared TLB by invalidating all entries of one of its own tasks. This

operation invalidates all the entries corresponding to the provided task and resource identifier or to a task of the resource requesting the operation. The R-ID value from the R-ID register of the requestor is used to qualify the operation.

[56] For an "invalidate all entry related to a Resource" operation, all entries corresponding to RID field 404 of the control word are invalidated. Note that all processors of a given megacell embodiment might not be allowed to invalidate entries other than their own. This provides, however, the capability to a master processor to free space from the shared TLB by invalidating all entries of another processor. The R-ID value from the R-ID register of the requestor is used to qualify the operation.

[57] For an "invalidate all shared entries" operation, all entries in the TLB marked as shared for the requester are invalidated. The R-ID register value limits the effect of this operation, as discussed above.

[58] For an "invalidate all entries of a task except shared entries" operation, all entries in the TLB for a task specified in the control word not marked as shared for the requester are invalidated. The R-ID value from the R-ID register of the requestor limits the effect of this operation, as discussed above.

[59] For an "invalidate all entries" operation, all entries in the TLB matching the R-ID of the requester are invalidated. For the master CPU, the operation invalidate all entry regardless of the R-ID. If all of the R-ID registers distributed in the system have the same value, then this operation invalidates all entries.

[60] For a "lock/unlock entry" operation, a control word is written providing the VA which needs to be locked/unlocked. This operation sets or resets lock field 304 in the selected entry. Restriction on R-ID applies as above.

[61] For a "lock/unlock all entry related to a task" operation, a control word is written providing the task identifier which needs to be locked/unlocked. Restriction on R-ID applies as above.

[62]   In the case in which an independent OS is running on each processor, each OS can initiate the above operations. In that case, these operations must be restricted to entries with a resource identifier (R-Id) belonging to the requester.

[63]   In the case of a single master OS, task and memory management can be viewed as unified, removing the need for an R-Id. The R-ID can be an extension of the task-ID. In an embodiment, in which the R-ID is hard-coded, the field R-ID in the TLB simply needs to be disabled (associated Valid bit is cleared) via a configuration control register. Disabling the R-ID is equivalent to having a single R-ID for all the system or for part of the system.

[64]   As mentioned above, a global control bit can be used in an embodiment to determine if all the above functions must be limited to the entry corresponding to the resource ID requesting the operation.

[65]   Although it is preferable to have the same page size for memory management on all processors, it is not mandatory. In a shared system, the TLB supports all page sizes of the system, in response to S/P field 306. Therefore, in a different embodiment, a TLB may support a different set of page sizes.

[66]   Table 5 also lists some additional operations that are provided which allow a software TLB handler to access the shared system TLB: Read TLB entry, Write TLB entry, Check and select victim TLB entry, and Set victim TLB entry. These are described in more detail below.

[67]   For a "Read TLB entry" operation, an entry in the TLB pointed to by the victim pointer is transferred into TLB entry register 330. The TLB entry register can then be read and analyzed by the software TLB handler. Again this operation might be restricted to the master CPU for security.

[68]   For a "write TLB entry" operation, the contents of the TLB entry register is transferred to a selected victim entry of the TLB.

[69]   The "check and select victim TLB entry" operation has multiple functions. Its first purpose is to determine an index value for the replacement of an entry. However, it can also be used to find out if an entry is already in the TLB. The

R_ID & Task_ID & VA fields of a corresponding entry are checked for a match against a proffered virtual address entry. If there is no match, then the victim pointer is positioned according to the chosen replacement algorithm. This replacement can be random, cyclic, etc. The second usage is to verify if a given page is present in the TLB. If a matching entry is found, the victim entry points to this matching entry, and a flag bit in the status register is set to indicate this condition.

[70]   The "Set victim TLB entry" operation allows the software TLB handler to select a particular entry as the next victim. This is useful to support certain lock mechanisms software replacement algorithms.

[71]   As indicated earlier, each control operation is performed by a state machine within TLB control circuitry 320 in response to writing to a selected memory mapped address. For example, for the operation "invalidate all entries related to a task", all entries with a matching task-id TAG are invalidated in response to a single command, including the shared TLB and the associated μTLBs. In the present embodiment in which the TLB is a fully associative memory, the operation can be done in one cycle or as a loop as most appropriate.

[72]   As mentioned above, control operation affect the shared TLB and the associated μTLBs for the various operations based on task-ID, resource-ID and shared bits. In an embodiment in which both uTLBs and TLB are fully associative, the flush and/or Lock/unlock can be done by the same command in the same cycle. But if the uTLB is fully associative and TLB is set associative, for example, a single command is still used, but the operation into the set associative TLB will be executed entry by entry by a HW loop. This will take longer time. If both the uTLB and TLB are fully associative there will typically be a single control block. If the uTLB is fully associative and TLB set associative, there may be separate control blocks 320, but the same command effects all of the control blocks. Alternatively, an embodiment may require sending copies of the operation command separately to each of the separate control blocks.

[73]   Figure 6 is a simplified block diagram of the TLB of Figure 3A and will now be referred to explain selective invalidation of an entry for a given task or resource, as listed in Table 5. Processor 2100(m) is representative of one or more requestors that access TLB 2130. A physical address bus 2104(m), resource ID signals 2106(m), and task ID signals 2108(n) are provided by each processor 2100(n) for each TLB request. Traffic controller 2110 provides request priority selection and sends the highest priority request to TLB 2130 using physical address bus 2104, resource ID signals 2106, and task ID signals 2108 to completely identify each request.

[74]   A task-ID field 302 and/or a resource ID field 301 stored as independent fields in the TLB TAG array is used to selectively invalidate (flush) all entries of a given task or a given resource (requester). A state machine within control circuitry 2132 receives a directive from a processor to perform an invalidation operation, as described above. The operation directive specifies which task-ID is to be flushed using format 400 or 410 (see Figure 5).

[75]   For operations which use task ID field 402 in the control word, state machine 2132 accesses each entry in TLB 2130, examines the task-ID field, and if there is a match that entry is flushed by marking the valid bits in its valid field 307 as not valid. Thus, a single operation is provided to flush all entries of a given task located in a TLB. As discussed above, in this embodiment, the TLB cache is made of several levels of set associative TLB and μTLB, and all levels are flushed simultaneously in response to a single operation directive command by accessing each entry sequentially in a hardware controlled loop.

[76]   For operations which use both task ID field 402 and R-ID field 404 in the control word, state machine 2132 accesses each entry in TLB 2130, examines the task-ID field and the resource ID field, and if there is a match in both the task ID and R-ID fields that entry is flushed by marking all valid bits in its valid field 307 as not valid. Advantageously, this allows discrimination of entries belonging to tasks from different resources that have the same task ID number. When the R-ID

TI-31357   - 28 -

valid bit is set, an entry is not flushed if its R-ID field 301 does not match the value provided on R-ID signals 2106. This operation only invalidates entries with a valid task-ID.

[77]   In a similar manner, the selective invalidation operation "Invalidate all entries related to a R-ID" is performed by examining the R-ID 301 field of each entry and if there is a match in the R-ID field that entry is flushed by marking its valid field 307 as not valid. This operation only invalidates entries with a valid R-ID.

[78]   Likewise, the selective invalidation operation "Invalidate all shared entries" is performed by examining the share field 303 of each entry and if there is a match in the shared field that entry is flushed by marking its valid field 307 as not valid. All entries marked as shared can be flushed in one cycle.

[79]   In the present embodiment, when shared entries are flushed, state machine 2132 ignores the task ID field since shared page entries may be used by different tasks having different task IDs. In an alternative embodiment, shared entry flushing could also be qualified by the task ID field. Alternatively, shared entry flushing could also be qualified by the task ID field, but only if the task ID valid bit in valid field 307 is asserted indicating a valid task ID value is in field 302.

[80]   Figure 7 is a simplified block diagram of the TLB of Figure 3A and will now be referred to explain selective lock/unlocking of an entry for a given task or resource, as listed in Table 5. Advantageously, in this multi-processor system with system shared TLB, an innovative scheme of adaptive replacement is provided for controlling the TLB on a task basis, as discussed above. In order to support such a function in the most optimized way, an adaptive replacement algorithm taking into account locked entries and empty entries is provided. TLB full signal 2240 is asserted when one or more valid bits in field 307 is asserted for each TLB entry location. TLB miss signal 2242 is asserted to indicate a miss occurred in response to a transaction request from processor 2100(m), which invokes a TLB handler as described earlier.

[81]   When the TLB is full with no locked entries, pseudo-random replacement based on a simple counter (Victim CNT) 2234 is used to select the victim entry. Another embodiment would be to keep a pseudo random replacement and to check the lock bit on a miss. If it is locked, signal 2244 is asserted and the victim counter is incremented further until a non-locked entry is found. This is done automatically by the control circuitry connected to victim counter 2234 so that response time of the TLB handler routine is not impacted.

[82]   When the TLB is not full, the victim counter is incremented until an empty entry is found. This is done automatically by the control circuitry connected to victim counter 2234 so that response time of the TLB handler routine is not impacted.

[83]   After a flush entry operation is performed, the victim "counter" is updated with the location value of the flushed entry and stays unchanged until a new line is loaded in order to avoid unnecessary searching.

[84]   An alternative implementation provides the capability to do the victim entry search instantaneously by providing in an external logic the lock and valid bit or by using a CAM, for example. In another alternative embodiment, a shift register and associated circuitry is used to point to the next location in the TLB that is either not valid or valid and not locked.

[85]   Figure 8 illustrates how a shared page entry is replicated for each task for different virtual address spaces. In this example, there are illustrated two tasks, referred to simply as task 1 and task 2. Each task may occupy several pages of virtual address space that are mapped to corresponding pages of physical address space. These various pages hold code, data, etc; however, for simplicity only one or two pages for each task are illustrated here for simplicity.

[86]   Task 1 has a page VA1 in virtual address region 900. Page VA1 is mapped to physical address space 910 at physical address page PA1. Task 2 has a page VA2 in virtual address region 902. Page VA2 is a shared page and is mapped to the same physical address page PA1. Also illustrated is a second page owned by

task 2 in virtual address page VA3. This page is not shared and is mapped to physical address page PA2.

[87]   Table 6 illustrates six entry locations of an example TLB that is loaded with entries for task 1 and task 2. The R-ID and attribute fields are not illustrated, for simplicity. The page size field S/P holds a size value of M, but different values can be specified. Entry 2 holds page VA1 of task 1, which is shared, as indicated by the shared bit S being set to "1." Entry 4 holds page VA2 of task 2, which is shared, as indicated by the shared bit S being set to "1." Entry 5 holds page VA3 of task 2, which is not shared, as indicated by the shared bit S being set to "0." In this case, each shared entry is replicated for each task because they are in different VA regions.

[88]   Advantageously, when either task 1 or task 2 is terminated and physical memory reclaimed, all shared entries can be expunged from the TLB by performing an "invalidate all shared entries" operation directive. In this case, an invalidate all shared entries operation will invalidate entry locations 2 and 4.

Table 6 – Example TLB for Tasks in Same VA Space

Entry #	Task ID	Vt	S	L	VA page	V	S/P	PA page
1						0		
2	Task 1	1	1	x	VA1	1	M	PA1
3						0		
4	Task 2	1	1	x	VA2	1	M	PA1
5	Task 2	1	0	x	VA3	1	M	PA2
6						0		

[89]   Still referring to Figure 8, in an alternative system embodiment in which the operating system does not use overlayed virtual address space, each task is in its own virtual address space. In that case, region 900 and region 902 would be disjoint and separate VA spaces. In the case of overlayed virtual address space, the TLB will need to be flushed each time a context switch occurs to change the execution thread from one task to a different task. Typically, the "active user process (or task)" is mapped on a defined VA range, which is the same for all user

tasks. Translation tables are modified at context switch to map the current user task in this range, and the OS tasks have another dedicated VA range whose translation does not need to be changed on context switches. In some OSes, MSBs of the VA can be used to identify the task ("process ID"), in order to reduce flushes of the TLB and caches.

[90] In the latter case, a task ID is not needed in the TLB since each task is distinguished by the virtual address space, as illustrated Table 7. In this case, separate virtual address spaces are distinguished by msbs of the VA page, for example. In this case, each shared entry is replicated for each task because they are in different VA spaces.

Table 7 – Example TLB for Tasks in Same VA Space

Entry #	S	L	VA page	V	S/P	PA page
1				0		
2	1	x	VA1	1	M	PA1
3				0		
4	1	x	VA2	1	M	PA1
5	0	x	VA3	1	M	PA2
6				0		

[91] In the case in which the TLB must be flushed for each context switch, an operation to "flush all entries of a task except shared" is useful. For example, when a task is completed or suspended, it's entries could be removed by this command, but any entries that were shared by a still active task would be spared. In another embodiment, entries for the OS could be marked shared. Advantageously, in this alternative embodiment, during a context switch, the entries relating to the OS and marked shared would not be flushed, for example.

[92] Figure 9 illustrates how a shared page entry is used by each of the sharing tasks in a single virtual address space. In this example, there are illustrated two tasks, referred to simply as task 1 and task 2. Each task may occupy several pages of virtual address space that are mapped to corresponding

pages of physical address space. These various pages hold code, data, etc; however, for simplicity only one or two pages are illustrated here for simplicity.

[93]   Task 1 has a page VA1 in virtual address region 1000. Page VA1 is mapped to physical address space 1010 at physical address page PA1. Task 2 has a page VA2 in virtual address region 1002. Page VA2 is mapped to physical address page PA2. Also illustrated is a virtual address page VA3 that is shared by task 1 and task 2. This page is mapped to physical address page PA3.

[94]   Table 8 illustrates six entry locations of an example TLB that is loaded with entries for task 1 and task 2. The R-ID and attribute fields are not illustrated, for simplicity. Entry 2 holds page VA1 of task 1, which is not shared, as indicated by the shared bit S being set to "0." Entry 4 holds page VA2 of task 2, which is not shared, as indicated by the shared bit S being set to "0." Entry 5 holds shared page VA3, as indicated by the shared bit S being set to "1." Note that the Valid Task ID (Vt) bit is set to 0 to cause the task ID field to be ignored. In this case, the shared entry is used by each of the sharing tasks in the same VA space; therefore only one entry is needed and the task ID field is ignored

[95]   Advantageously, when either task 1 or task 2 is terminated and physical memory reclaimed, all shared entries can be expunged from the TLB by performing an "invalidate all shared entries" operation directive. In this case, the invalidate all shared entries operation will invalidate entry location 5.

Table 8 – Example TLB for Tasks in Same VA Space

Entry #	Task ID	Vt	S	L	VA page	V	S/P	PA page
1						0		
2	Task 1	1	0	x	VA1	1	M	PA1
3						0		
4	Task 2	1	0	x	VA2	1	M	PA2
5	xxxx	0	1	x	VA3	1	M	PA3
6						0		

Digital System Embodiment

[96]   Figure 10 illustrates an exemplary implementation of an example of such an integrated circuit in a mobile telecommunications device, such as a mobile personal digital assistant (PDA) 10 with display 14 and integrated input sensors 12a, 12b located in the periphery of display 14. As shown in Figure 10, digital system 10 includes a megacell 100 according to Figure 1 that is connected to the input sensors 12a,b via an adapter (not shown), as an MPU private peripheral 142. A stylus or finger can be used to input information to the PDA via input sensors 12a,b. Display 14 is connected to megacell 100 via local frame buffer similar to frame buffer 136. Display 14 provides graphical and video output in overlapping windows, such as MPEG video window 14a, shared text document window 14b and three dimensional game window 14c, for example.

[97]   Radio frequency (RF) circuitry (not shown) is connected to an aerial 18 and is driven by megacell 100 as a DSP private peripheral 140 and provides a wireless network link. Connector 20 is connected to a cable adaptor-modem (not shown) and thence to megacell 100 as a DSP private peripheral 140 provides a wired network link for use during stationary usage in an office environment, for example. A short distance wireless link 23 is also "connected" to ear piece 22 and is driven by a low power transmitter (not shown) connected to megacell 100 as a DSP private peripheral 140. Microphone 24 is similarly connected to megacell 100 such that two-way audio information can be exchanged with other users on the wireless or wired network using microphone 24 and wireless ear piece 22.

[98]   Megacell 100 provides all encoding and decoding for audio and video/graphical information being sent and received via the wireless network link and/or the wire-based network link.

[99]   It is contemplated, of course, that many other types of communications systems and computer systems may also benefit from the present invention, particularly those relying on battery power. Examples of such other computer systems include portable computers, smart phones, web phones, and the like. As

power dissipation and processing performance is also of concern in desktop and line-powered computer systems and micro-controller applications, particularly from a reliability standpoint, it is also contemplated that the present invention may also provide benefits to such line-powered systems.

[100]   Fabrication of the digital systems disclosed herein involves multiple steps of implanting various amounts of impurities into a semiconductor substrate and diffusing the impurities to selected depths within the substrate to form transistor devices. Masks are formed to control the placement of the impurities. Multiple layers of conductive material and insulative material are deposited and etched to interconnect the various devices. These steps are performed in a clean room environment.

[101]   A significant portion of the cost of producing the data processing device involves testing. While in wafer form, individual devices are biased to an operational state and probe tested for basic operational functionality. The wafer is then separated into individual dice which may be sold as bare die or packaged. After packaging, finished parts are biased into an operational state and tested for operational functionality.

[102]   The digital systems disclosed herein contain hardware extensions for advanced debugging features. These assist in the development of an application system. Since these capabilities are part of the megacell itself, they are available utilizing only a JTAG interface with extended operating mode extensions. They provide simple, inexpensive, and speed independent access to the core for sophisticated debugging and economical system development, without requiring the costly cabling and access to processor pins required by traditional emulator systems or intruding on system resources.

[103]   As used herein, the terms "applied," "connected," and "connection" mean electrically connected, including where additional elements may be in the electrical connection path. "Associated" means a controlling relationship, such as a memory resource that is controlled by an associated port. The terms assert,

assertion, de-assert, de-assertion, negate and negation are used to avoid confusion when dealing with a mixture of active high and active low signals. Assert and assertion are used to indicate that a signal is rendered active, or logically true. De-assert, de-assertion, negate, and negation are used to indicate that a signal is rendered inactive, or logically false.

[104]   While the invention has been described with reference to illustrative embodiments, this description is not intended to be construed in a limiting sense. Various other embodiments of the invention will be apparent to persons skilled in the art upon reference to this description. For example, in another embodiment, the TLB may be limited to a single processor and not shared, or it may include only a single level without μTLBs.

[105]   In another embodiment, the TLB may be controlled by other means than a state machine controller, such as directly by an associated processor, for example.

[106]   In another embodiment, there may be several distinct MMUs with associated TLBs, wherein certain of the TLBs may include aspects of the invention and certain others may not.

[107]   It is therefore contemplated that the appended claims will cover any such modifications of the embodiments as fall within the true scope and spirit of the invention.